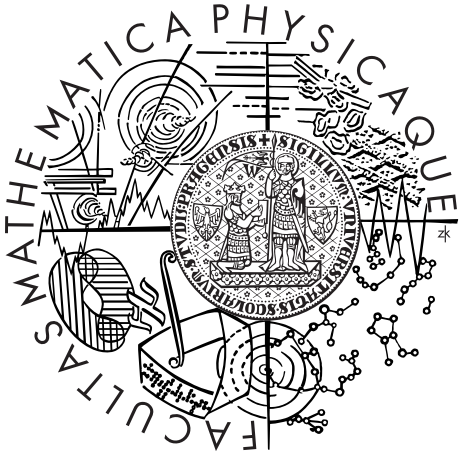
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Peter Júnoš

Extending Java Performance Monitoring Framework with Support for Linux Performance Data Sources

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Lubomír Bulej
Study programme: Computer Science
Specialization: General Computer Science

Prague 2012

I would like to thank to my consultant Lubomír Bulej, who was leading this thesis for his support and for his explanations of basics of the Java Performance Measurement Framework.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Extending Java Performance Monitoring Framework with Support for Linux Performance Data Sources

Autor: Peter Júnoš

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Ing. Lubomír Bulej, Ph.D

Abstrakt: Java Performance Measurement Framework (JPMF) je framework, ktorý sa dokáže včleniť do programu a získať štatistiky o výkonnosti počítača v zadaných sledovacích bodoch. Hlavnou nevýhodou súčasnej implementácie sú chýbajúce senzory, ktoré by dokázali merať výkonnostné štatistiky harddiskov, procesoru, pamäti a sieťových rozhraní.

Linux neposkytuje jednotné funkcie na prístup k týmto výkonnostným štatistikám. Tie je možné získať z virtuálnych súborových systémov, syscallov a rozhrania Netlink. Cieľom tejto práce je rozšírenie JPMF tak, aby poskytoval meranie spomínaných výkonnostných štatistík tak, ako sa to v Linuxe.

Klíčová slova: meranie výkonnosti, JPMF, Linux, štatistiky z netlink, štatistiky z procfs

Title: Extending Java Performance Monitoring Framework with Support for Linux Performance Data Sources

Author: Peter Júnoš

Department: Department of Distributed and Dependable Systems

Supervisor: Ing. Lubomír Bulej, Ph.D

Abstract: Java Perfomance Measurement Framework (JPMF) is a library, that is able to hook into program and gain performance information in given watchpoints. Notable drawback of currect implementation are missing sensors, that would be able to measure performance statistics related to storage, CPU, memory and network interfaces under Linux.

Linux does not provide unified way of accessing such performance statistics. They can be accessed using virtual file systems, syscalls and netlink interface. The goal of this work is extending JPMF, so that it will provide measurement of mentioned performance statistics in a Linux-specific way.

Keywords: performance measurement, JPMF, Linux, netlink stats, procfs stats

Contents

1. Introduction

Programming is often full of decisions. Programmer decides, which implementation will be better based on some information about each implementation. With growing amounts of data, performance does not lose its importance, despite the new inventions in the field of hardware. It is also a notable aspect of software quality and user experience, which can be observed in reactions of new Czech vehicle registry users[16].

Approach based on theoretical system analyses give desired results only when calculated asymptotic bounds are not very close to each other and when size of their input is big enough to manifest the difference between asymptotic complexities. This approach can be used when excluding bad algorithms. As an example, we can exclude a a program with exponential running time when there is another running a polynomial running time. Complexities, that are close, can give incorrect results. Example of such results would be comparison of quicksort and heap sort, that have the same asymptotic complexities, except worst case, where the heap sort wins with $n \log n$[17, p148] compared to n^2 complexity of quicksort[17, p121]. In real tests, results are swapped – quicksort is faster than heapsort. Such errors are caused by differences between average cases and worst cases, by lower sizes of real-life data and by omission of data related to caches and their efficiency. Solution of the last mentioned problem could be an introduction of cache-oblivious model. Cache-oblivious model can lead to valid results for inputs, that are big enough. Comparison of Quicksort and Funnelsort by Brodal et al[15] describes expected efficiency when input was almost as big as RAM.

Therefore, theoretical analysis can give valid results, but only under specific circumstances. Benchmarking is a way to compare programs under load without possibility of forgetting any aspects of benchmarked program, with respect to the real size of user inputs and other unmentioned differences.

Nonintrusive benchmarking provides partially useful information, but does not give any idea about performance of individual functions and performance critical code sections.

We would like to get benchmarking, that can be turned on and off whenever performance begins to be an issue without interrupts in provided services.

Java Performance Measurement Framework (JPMF) is a framework for collecting performance data from system level[2]. The data are collected in specific watchpoints called Events, which are described in a separate configuration. Data to be harvested are acquired on-demand as expressed in configuration. The data are specified as sensors where one sensor means one type of information to be acquired. Therefore, sensor can provide CPU time used since reboot, HDD accesses by Java, RAM usage or network statistics. Sensors will be discussed further in Chapter 1.1.2.

User specifies sensors to be acquired and events, when should be the data acquired and gets sampled data either directly or in the form of statistics.

Using these stats, one can see critical points of program together with its resource consumption, that opens a way to do selective speed optimalisation.

1.1 JPMF architecture

JPMF is divided into subsystems, that are divided to two parts as imaged in Figure 1.1. Application-driven subsystems are Event Sources, Performance Data, Event Processing and Data Storage. Although the number of parts is high, we do not have to explain all of them, because, our work will extend only one Performance Data Access Subsystem. Next, we will discuss parts of Performance Data Access (Subsystem) and also mention its important parts.

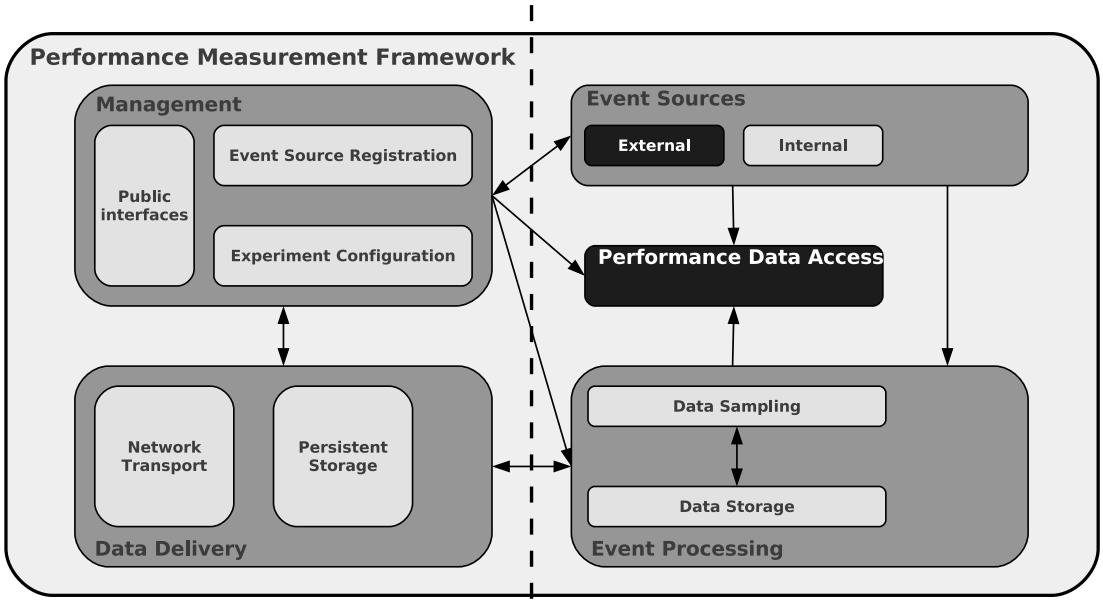


Figure 1.1: Design of JPMF (image by Bulej[1])

1.1.1 Performance Data Access Subsystem

Perfomance Data Access Subsystem is a part of the framework, that can run itself, without any helper libraries. Its main goal is providing abstraction on the top of various sources, that will provide getting performance data from various sources. There are multiple sources of data, that are ranging from sources provided by CPUs, going thru sources provided by an operating system and finishing in data sources found in applications [1, p32].

On the hardware level, we have `x86` intruction, called `RDTSC`, that is provided since Pentium. This instruction can be used to get number of CPU cycles since reboot and can be therefore usable in performance measurement[18].

Multiple performance data are provided by operating systems. Windows provides `Windows Management Instrumentation` and unsupported syscalls[3], Solaris provides `kstat` and syscalls and Linux provides virtual file systems, netlink and syscalls.

Additionaly, as caused by different running environments, there are multiple ways to access data, that is said to often require APIs available only in in low-level languages [1, p33].

Performance Data Access Subsystem relies on data sources and time sources, that should provide more types of performance and time data, such as I/O stats, CPU usages, netwok stats and even more data.

Time sources are providers of time from various sources, which differ in granularity and resolution. Because they are implemented and do not have anything to do with our thesis, we will not discuss it further.

1.1.2 Sensor

Performance data are provided in form of sensors, where one sensor means one kind of performance data that can be acquired, such as "harddisk operations" or "cpu cycles spent" or any different kind. Sensor does not neccessarily contain only one value, because there can be more parts of hardware providing same performance data.

To solve the mentioned issuse, JPMF authors decided to introduce idea of sensor instances, that could be easily explainable as "sensor parts". This approach still provides one sensor for one kind of performance data and extends it in a way, that if multiple performance data of same kind are accessible, they should be acquired and saved to different instances of same sensor. In an example, we can provide sensor like "I/O operations" with instances "first HDD" and "second HDD". However, singleton, that is a sensor with one instance is still supported. Singletons are suitable especially for performance counters, that do not have multiple instances by design, such as counts of RAM operations, where RAM is understood as one flat space.

This solution is perfectly structured, that allows better user interface and his orientation among various sensors. The most important is, that it is also completely platform independent and it will not affect portability of whole framework.

Sensors provide more data than just one or few numbers, but it is not important now and it will be described in the chapter Design.

Sensor naming

Performance Data Access Subsystem is not able to provide naming for all Data sources. Since each Data source creates names of sensors itself and to avoid collisions, each Data source gets its own namespace under URL schema:[1, p58]

```
[sensor://]<datasource>/<group>/<sensor>[#<instance>]
```

where instance is not present for the sensors, that are singletons. We have will describe the concept of data source in the following chapter – for now, the important part is, that is uses string identification, that is used here as a part of URL schema. Group is only a logic way of grouping similar sensors[1, p58], like grouping all harddisk stats into one part of namespace. Groups reduce flatness of address space, while allowing easier reading and filtering.

1.1.3 Data Source

Data sources are classes containing various sensors, that are acquired using same basic method. We can have a data source providing performance data, that were acquired from virtual file systems, by syscall, from `WMI` or from `kstat`.

We are introducing them, because their names are neccessary in sensor identification string and they are the parts of whole framework, that interact directly with the remaining of the framework.

1.2 Goals of this work

JPMF does not yet support reading of broad spectrum of provided Linux performance counters. Therefore, main goal of this work is implementing data sources, that will provide reading of Linux performance counters describing disks, RAM and CPU and will read its data from virtual file systems and using native calls. Data sources will blend with JPMF, they will also use native calls and will provide an ability to be configured without Java code editation.

2. Analysis

We would like to describe, how can be the performance data harvested with advantages and disadvantages of each approach and then, select the best way of doing it.

2.1 Getting performance data

Performance data on Linux can be read from files and by using specific syscalls.

First-to-be-mentioned advantage is, that native code provides numbers in a binary format, ready to be returned to Java, which reduces overheat caused by converting numbers to text by Linux kernel and then back to numbers by the framework.

The main disadvantage of using the native code is a necessity of its external compilation and, later, loading native code to JVM. As a result, we are forced to use non-Java code, that is not managed by Java and could cause, in case of bug, instability of the whole framework without ability to catch it like Java exceptions.

Last approach in getting performance stats is writing custom kernel module or configuring kernel. We will not do thing, because we are not yet skilled enough to program kernel modules and it is not necessary according to our calculations.

There are two basic ways of getting performance information in native code: by specific syscalls or by netlink interface.

2.1.1 Syscalls

We have found 4 syscalls, that are able to provide performance data. All except one get kernel performance data kernel information in a standardized approach, as suggested by Single Unix Specification (SUS)[5]. Standardization means stability of calls across different versions of Linux and also across Unices, one of which is Solaris. Therefore, Linux sensors using syscalls specified by SUS could also work on Solaris. Syscalls are not a sufficient solution for full Solaris suppport, because Solaris provides richer `kstat` interface containg structured binary data from kernel on one place. Despite of the mentioned facts, syscalls in SUS can be used for measuring performance data on Solaris with the advantage, that it would not require separate implementation.

Another advantage could be an amount of overheat, because one system call is enough to get multiple performance data, contrary to virtual files containing single value, where there are at least 2 syscalls required to get data. However, this computation is only theoretical and therefore can be incorrect, as there are various variables, that affect the speed.

Each system call2.1 returns its data in customized structure and, because of that, necessarily requires specific native code for each syscall, so implementing generic access from Java results of system calls cannot be done. Structure for syscall `getrusage()` can be seen in Figure 2.1.

There are also syscalls such as `sysinfo()`, that are not described in Single Unix Specification - we will use them in a similar way to that described with

`getrusage()`	gets resource usage related to CPU time used, memory and process management; in SUS
`times()`	gets system and user time for current process; in SUS
`statvfs()`	gets harddisk stats for given files; in SUS
`sysinfo()`	gets overall system stats, mainly concerning memory and system load; not in SUS

Table 2.1: Linux syscalls able to get performance data

```
struct rusage {
   struct timeval ru_utime; /* user CPU time used */
   struct timeval ru_stime; /* system CPU time used */
   long   ru_maxrss;        /* maximum resident set size */
   long   ru_ixrss;         /* integral shared memory size */
   long   ru_idrss;         /* integral unshared data size */
   long   ru_isrss;         /* integral unshared stack size */
   long   ru_minflt;        /* page reclaims (soft page faults) */
   long   ru_majflt;        /* page faults (hard page faults) */
   long   ru_nswap;         /* swaps */
   long   ru_inblock;       /* block input operations */
   long   ru_oublock;       /* block output operations */
   long   ru_msgsnd;        /* IPC messages sent */
   long   ru_msgrcv;        /* IPC messages received */
   long   ru_nsignals;      /* signals received */
   long   ru_nvcsw;         /* voluntary context switches */
   long   ru_nivcsw;        /* involuntary context switches */
};
```

Figure 2.1: Output structure of `getrusage()` copied from getrusage manual page

syscalls mentined in Specification. The major disadvantage of these syscalls is, that they probably will not work on other Unix-like systems.

2.1.2 Netlink interface

Netlink interface is recommended network-like mechanism for communication between kernel and userspace applications. On its packet can be seen in Figure 2.2.

Accessing data exposed by kernel through Netlink can be done in a partially unified way. It is only partially, because part of Netlink is a protocol, that is generic enough and that could be directly exposed to Java similarly to access to network. On the other hand, communication itself is not standardised and has to be done differently for different data, that would be retrieved using Netlink.

We should also mention, that there are only few documented kinds of information, that can be read using netlink. This reduces importance of generic access in favour of customized code for each call.

We have to decide, whether we will use a library, like there is libnl, or we should use functions directly.

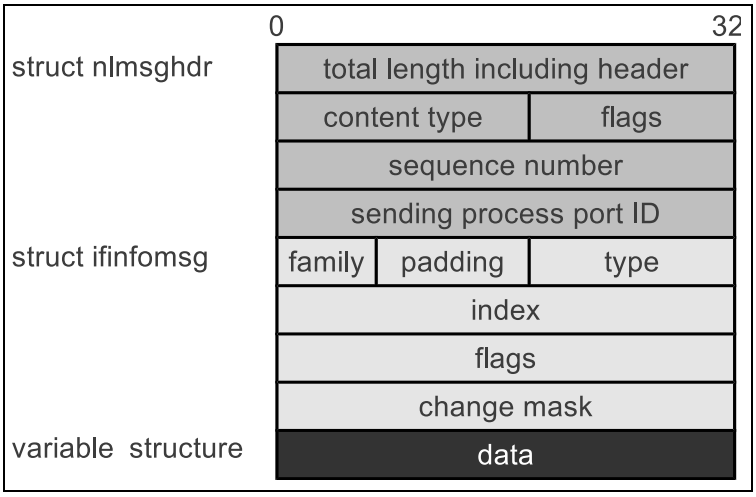


Figure 2.2: Common structure of netlink packet

Direct use of networking functions bears less abstraction and possible simpler access to kernel. On the other hand, `libnl` helps mainly with a formal side of communication. It can construct a packet for a user, without explicit specification of the structure alignment and without counting bytes to insert correct size to a packet. Moreover, `libnl` cares about correct padding, that is so low level detail without any advantages if user cared for it.

Last but not least, `libnl` provides very simple debugging feature – setting environment variable is enough for libnl to output whole communication and even try to guess fields of received packet.

We have decided to use `libnl` mainly because of the debugging possibilities. Same expresing power without the necessity of caring for low level details and callback system are nice additional benefits. Even if these benefits were not present, wide debugging capabilities would be enough to use `libnl`.

2.2 Virtual file systems

Virtual filesystems (VFS) are special types of file systems, that do not provide data saved in non-volatile memory like traditional file systems, but expose internal parts of Linux kernel and provide actual data, whenever read. Despite this difference, virtual file systems still provide files in directory structure. Files can be manipulated using same syscalls, that are used to access and change traditional files[1]. Virtual file systems available in vanilla Linux kernel are sysfs and procfs, therefore we will sometimes mention these two public implementations in place of VFS.

Files in virtual file systems provide not only various system and process information, but also a configuration of Linux system internals via special files. We are not going to configure kernel and therefore we are not interested in this part and writing to VFS.

Contary, we are interested in system and process information, that could be used in JPMF as sources of performance data.

[1]There is one difference between virtual and traditional file systems - reported size of files is zero for all virtual files

Major advantage of using them to gather performance data is, that all information are in virtual files that can be read using Java functions without native libraries. Therefore, this part is not required to contain native code with all its disadvantages, such as harder error checking.

Despite the advantages, reading files from file systems can be more resource consuming than getting them via native means. Each reading of a file requires at least two system calls - one is for opening file or seeking to begining and the other one is for reading. There is also an other implementation detail, that slows down reading in our framework - contents of the files are made to be easily readable by a human, that implies they are in ASCII and have to be converted to binary before using in our framework. Moreover, human readability was sometimes much more important than machine parsing - we can take look at `/proc/net/dev` or `/proc/net/wireless`, which contains tables drawn using ASCII characters, as can be seen in Figure 2.3.

```
Inter-| sta-|   Quality        |   Discarded packets               | Missed | WE
 face | tus | link level noise |  nwid  crypt   frag  retry   misc | beacon | 22
  eth1: 0000    5.  -256.  -256.       0      0      0      0      0        0
```

Figure 2.3: Example of `/proc/net/wireless`

Note that reading files becomes much harder, as structure of files differs across sysfs and procfs. Thus, we analysed types of files, that are found in

2.2.1 Single value files

The most prevalent group of files, which are encouraged to be used in sysfs[7, Documentation/hwmon/sysfs-interface], are files, that contain only one number, that is also the value to be read. Such files do not require special handling except support for reading one number. Contents of one single value file can be seen in Figure 2.4.

```
3112635255
```

Figure 2.4: Single value file - `/sys/fs/ext4/sda5/lifetime_write_kbytes`

Values in these files are partially or fully identified by filename, where are the files located. Because we have a filename, we should provide user an ability to specify multiple filenames at once and say, which parts of filename should be interpreted as a name of sensor or instance.

We should mention, that reading these files carry increased resource usage related to file desciptors, because one open file descriptor for single value file allows getting data for only one sensor instance. We could close the file after every use, but such usage would need more syscalls - for opening, reading and eventually closing file.

Fortunately, there are also another resources, such as memory or CPU cycles, that are spared when values are read from single value files. Files containing single value do not require getting or parsing other values (there are no other values), when they are not required. Parser just gets the result without having to search for it.

We will certainly support single value files, not only because of their count, but also because their format is so readable and easy to understand.

2.2.2 Files containing name and value

Different group of files, are files where every line contains name for value, possibly separator and value, possibly followed by skippable (and stable) unit. These files are interesting, because they provide multiple values, that are at least partially described by name. Since name of each value is provided by containing file, these files can be configured more easily than files of other formats.

Files with names and values are also a good ballance between resource usage caused by number of open files and overheat caused by reading unnecessary data.

Part of such file can be seen in Figure 2.5.

```
MemTotal:        3891600 kB
MemFree:          547932 kB
Buffers:          206704 kB
Cached:          1159840 kB
SwapCached:            0 kB
Active:          1986148 kB
Inactive:        1002752 kB
Active(anon):    1623552 kB
Inactive(anon):     2140 kB
```

Figure 2.5: Part of `/proc/meminfo`

2.2.3 Files containing multiple values

Another group are files, that contain multiple values without description and user has to know meaning of values. Part of these files have stable format, that means format (number of lines or order or count of values) will not change even during reboots. This format will be supported by configuration, because it can be described in a generic way.

There are also files, that change their format when ran on different computers or with different hardware - we will support it only partially, if user provides description of file for current hardware.

Other file types are not going to be supported — this has multiple reasons. One is difficulty of creating easily understandable format string for such files and another are low count of this files and, followingly, not enough files to get idea of generic implementation. Despite of our decisions not to support such files, users can write their own parsers (or even whole probes or data sources), that will support every possible, even future, format.

To imagine contents of simple MultiValue files, you can look at Figure 2.6.

```
   72725    80128  3231543  4128404    30502    92476  5714176  5640280        0
   432436  9769080
```

Figure 2.6: Example of `/sys/block/sda/stat`

2.2.4 Dynamic and binary files

Last group of files are dynamic and binary files. These files change their structure (such as line count) based on external events, or these files don't contain plaintext. These can be read using specific readers, but we won't provide such readers in our framework. These files are also all files, that have instances on separate lines - such files change in case of change of instances.

2.3 Configuration

The main reason why we need a configuration is, that files in VFS change and we would like to provide a possibility to reflect these changes without changing our code. The next reason is, that framework requires multiple labels, called Descriptors, some human- and some computer-readable, that should be provided to user. These labels compiled in Java would not lead to readable code and furthermore, leads to unnecesary code, that have to be parsed by Java compiler and thus slowing down compilation.

2.3.1 Functioning as a data holder

Configuration should be made in an easily expandable way. Reading and editing source code repeatedly or adding custom code for each sensor like it is done by Munin does not satify these requirements, so we have decided to use separate configuration files.

Our configuration should be computer-readable but also sometimes editable by human, able to hold information about sensor descriptors, filetypes and files, that will be read during parsing.

We could use binary format, that would be effective, but it would not be human-readable not human-editable.

We think format should be well known, which will make configuration easier. There are two well known formats satisfiing this condition, as they are wide spread. One of them is INI, whose disadvantage is being not enough strict. Another format is XML, that is strict enough to catch mistakes in typing and to be verified easily using external tools. We know, that easily usable parsers of XML exist and satisfy all our needs.

Because format can change frequently, as well as file type of configuration, configuration will have to be separated from the rest of the framework by a separate class to "catch" all changes not affecting functions.

2.3.2 Building simple configuration

We have noticed, that long parts of paths or repeated parts for multiple instances should be often inserted one after another. Requiring user to enter whole path for each file is bug prone and also slow to enter. Moreover, location of procfs can differ from the wide spread location in /proc - it is not even hard to mount procfs anywhere. If a movement would occur, either by developers or by moving JPMF to different computer, user would have to rewrite all paths related to procfs, while possibly introducing further bugs. Luckily, sysfs is on the opposing side, that has

stable mountpoint `/sys` [7, Documentation/sysfs-rules.txt], but we still need a simple solution for the repetition, while entering data about procfs.

We have decided to introduce variables to solve this problem. Variable is a named expression, that translates to any number of values. Configuration parser should automatically translate all path choose all values, that lead to paths containing files and loads files from these paths. Variable should allow being defined using another variables, which could make configuration easier.

2.4 Related work

Programs designed for a performance measurement were already described[1, p10], but there are even more approaches to measure and provide performance data, that differ in what they do with data.

If there was a work, that provides comprehensive access to sensors in an unified way, we could build a bridge, that would provide all data to the framework.

There are complete frameworks and simple scanners of actual CPU or memory usage, mainly connected with regular monitoring and alerting user, if value is too big or too small. Such monitoring could provide only data-getting interface, that would be used by our program. We would not like to use whole source, that provides unwanted regular access without binds to checkpoints, where data will be measured.

One system for a system-wide monitoring is Munin[13], which is expandable by downloadable plugins, and provides an idea, where are the appropriate files located in physical filesystems – procfs and sysfs. Each plugin gives the idea about different part of virtual files system. Munin's main disadvantage is, that it does not provide unified access to performance data. Moreover, every plugin uses custom ways of getting data, that is not expandable in an easy way.

Another system is Ganglia[14], that is used by WikiMedia as a monitoring tool. It provides unified access to data, that is not provided by Munin, but this access is closely bound to Ganglia itself, without possibility of using its code without using and copying big parts of the remaining program.

Reporting values outside certain range. Another similar systems are collectd and nagios. Because the programs and their modules are opensource, they can serve as a documentation for files in virtual file systems, that are not documented or as a supplementary documentation for documented parts.

LeWYS[19] could be an interesting project to support , but it is still in Alpha stage and not developing, because last message came to the mailing list in April 2009 and all activities ended in March 2007. We do not want to implement product not developed actively, because there is a risk, that no new bugfixes will be published.

BEEN[20] seems to be live project, but simple source code analysis of getting performance data revealed, that BEEN misses more generic parsers for other files. Therefore, we will not make a bridge to this project in place of coding our own performance data gatherer.

2.5 Summary

Linux provides wide range of sources providing performance data, but all sources cannot be accessed using one global interface. We have identified types of files within VFS based on their content, with their significance based on the count of files with described structure.

2.5.1 File types

As a part of analysis, we have found all files, that changed after wide-ranged system load (including network traffic, HDD reads and writes, memory allocation and partial freeing) and, as such, are good target of our measurements.

Files, that did not change are not interesting, because they are not good for measuring performance. Their main idea is, in almost all cases, holding static configuration or configuration, that is static during one computer boot or configuration, that changes only when new device is plugged or old device is removed.

We have used a simple script to find all files, that change between different invocations, with heavy system load between invocations and different load during in the first and in the second run.

This gave us almost all files we would like to read, but, unfortunately, mixed with unneeded garbage, that will be never needed.

Results are shown in Table 2.2.

We should mention, that not all files are worth of getting support in JPMF. We will want to measure only data, that have something to do with performance and change without user doing explicit action. This means, we won't support getting information, that change only during reboot or during use of pluggable devices.

Other files and reasons for their inclusions or exclusions are mentioned further in the text.

Some files will not be supported because of their meanings:

ATI specific files - files, that were located in subdirectory named ATI and are probably related to graphic subsystem. We are not interested in reading and providing them, because they are available on small fraction of platforms without guarantees.

Slab objects files related to allocator - we have found no description nor usable values, so we have discarded them.

Other not wanted files containing mainly enumeriations of all kinds, such as memory maps or network connections and files not required current time, current shell command - we have no idea, what should be measured here, so we are going to ignore them.

Binary files - files containing binary data, that will not be parsed as we have found no description for them.

`SingleValue` files are the simplest and the most frequent of all file types, that should be easy to support and thus, we will definitely support it in our thesis.

File	File count
Single value	111
Name value	28
Multi value – values without description with skippable data	12
Multi value with unstable format when ran on different computers	8
Name value with multiple instances in different sections of file	4
Name value with more Instances on 1 line	2
Multi value with instances on different lines	2
ATI specific	5
Slab objects	107
Other not wanted	66
Binary	4

Table 2.2: Counts of changed files during analysis - below line are files, that are certainly not interesting

`NameValue` files are still frequent enough and also provide descriptions, which means easy parsing together with easy maintainability. There is still possibility, that some values will be added to files – only `NameValue` files are backward and also forward compatible. They provide names and values almost independently of file format and addition of new value does not have to be approved by user, as well as removal of old value. Such approach reduces amount of time required to maintain parsers, because only a big change (delimiter of whole file format) would need touch of maintainer.

`MultiValue` files are not very frequent, but we will support them mainly because they can be easily expanded to read almost all remaining files and such reading would be correct, although it would require additional effort from maintainter. The main drawback of these files is, that they affect portability, because their format is changing based on CPU count or other hardware components.

Any other files or their configuration would be hard to understand and could not be tested in area, that is wide enough (4 files with almost all info redundant is not enough to support it). Other files could be parsed with per-file custom probe, that would be much easier programmable and much more effective than making generic solution.

Reading of all these files should be described in a configuration to be easily changeable and readable for the maintainter, preferrable in XML, with some attention given for possible transition to another formats of configuration.

We have also described performance data, that can be acquired using native methods. Because using native code means unevitable modification of Java code, we will provide a data source, that will not be forced to be easily configurable thru same configuration. This data source will get its data using netlink and system calls.

2.5.2 Goals of the thesis revisited

Now we have concrete requirements to achieve the goals of the thesis and can closer specify requirements from previous chapter.

We would like to divided our thesis to two basic parts - getting performance

information from files and getting it using JNI. This division is based on different approach, that had to be used. Files have their structure and that structure can be usually described in generic way. This offers unified way to access their contents and extend framework by describing new files, that can, possibly, appear in new Linux kernels. On the other hand, every group of performance data gathered using JNI needs its own native code. This means user will have no chance to extend it, unless he will implement the missing part and compile the framework with it.

To sum up, we would like to fulfill requirements described by the goals:

- Build a datasource reading data from files containing single value, name and value with possible description and multiple values of same type
- Build a datasource reading data using native calls implemented in low level language. Low level language will get data using syscalls and Netlink interface, while providing interface prepared for future expansion, when new syscall will appear.
- We will require expandable and maintainable framework, that will require configuration for file data sources; this configuration will not be required, when reading data from native datasource

3. Design

Based on the summary of analysis, we have decided to make design, that will consist of two data sources. One data source will provide reading native data and the other will provide reading of data contained in files. The main reason of following the division is separating parts loaded natively, especially for systems, that do not have native library compiled, or such library cannot be loaded. Therefore, data sources reading performance data from JNI and from files should be completely independent on a logic level, but they can surely share code on programming level.

One goal has also suggested expandability and scalability and therefore, we have divided both data sources to several layers, that can be seen in Figure 3.1. These layers will be independent with the minimum size of data exchanged. Further, the scalability will be one of the main aspects, because we have already decided to divide data sources.

We will describe layers of our work going from the uppermost parts going top-down, from most general layer made mainly to connect our thesis to the remaining of the framework to lowest laying layer, that read raw data provided by kernel either in virtual file system or thru native calls.

3.1 File data source

File data source will be divided to multiple layers, that do not differ from architecture mentioned int Figure 3.1.

3.1.1 Data source

Data source is the main part of our thesis, that should integrate directly with the remaining of the framework. It will provide sensors with their groups and descriptors, as it is defined by `DataSource` and `DataSourceDescriptor` interfaces to blend with the framework completely.

It solves the scalability goal from analysis, as it is responsible for reading configuration. The best scalable solution was reading configuration here, because we do not want configuration on a very low level; nor we want configuration to be read repeatedly. Giving such responsibility to the highest layer gives such guarantee. Part of configuration is provided directly with descriptors and therefore no harm will be made by our decision.

Data source is divided to several parts - readers used to read simple data from prepared buffers called `OctetBuffer`s; parsers, that are able to describe data provided by readers by providing meanings of measured values together with values; probes, that will open files and provide octet buffers and will be also holders of sensor groups and data sources to group similar approaches of access to performance data.

3.1.2 Probes

Data sources in JPMF provide data using probes, as groups provided by data sources provide `ProbeContext`s, that will be used to measure data.

Originally, probes are only groups of sensors, that use same resources, such as open files, that is scalable, but it can be made even better.

We have extended this basic idea and decided to make one-to-one relation between sensor groups and probes to make it easier to understand and also easier to implement. This decision takes us a possibility to make universal sensor grouping, that would provide one group for logically related sensors, without relation to their source. However, we have already mentioned counter-measure called virtual data source, that is used to make naming uniform between operating systems[1, p59]. If we require uniform naming, then the one-to-one relation will also save part of system resources, that would be required to make two mappings in a row. Such decision reduces barriers in scalability, that could appear, if someone added multiple mappings, that would use the resources pointlessly.

Moreover, this design allows creator of Virtual data source to see groups associated with same resources and select sensors from lowest possible number of groups, that will reduce resource usage.

Probe context

The main idea of Performance Data Access is getting values and their storage, which is in this thesis called measurement. This can be mistaken with sampling, that is only one part of whole process – data acquisition, that could mean only saving raw performance data to prepared buffer, without parsing itself. It begins with preparation and is finished by decoding, that parses acquired data and saves it to prepared classes, called `ValueHandles`. These functions are called in a row in single-threaded environment, but in multi-threaded environment, some parts can be executed in different thread[1, p61].

The sensor naming described in the Chapter 1.1.2 requires a search for appropriate data source by its name and data source looks for the matching group and sensor. These lookups should not be done everytime when accessing sensor, if we would like to reduce an usage of the system resources. Probe context is a solution provided by JPMF – the framework requests sensors for given names and receives the probe context, that provides direct access to all sensors.

All measurement functions are accessed thru `MeasurementContext` interface provided by `PerformanceDataManager`[1, p62].

`ProbeContext` is only a restriction of `MeasurementContext` restricted to functions able to measure data. Its main idea is exposing `Probe`, that is otherwise internal structure directly providing data, that will be internally acquired together[1, p68]. Therefore, it is a good idea to have one `Probe` for each chunk of acquirable data, where one chunk of data can contain more values.

Such approach allows refreshing one probe only once during sampling and allows saving system resources, that would be used by repeated refreshes.

3.1.3 Parsers

Parsers (depicted in Figure 3.2) are classes, that provide understandable data together with names of values contained in data. Parsers and lower classes cannot be seen by the remaining of the framework, because they provide everything thru `ProbeContexts`. These data are provided automatically, after calling `sample()` on related parser, that places requested values to prepared `ValueHandle`s.

The main goal of each parser should be providing naming and data itself while skipping unnecessary data, provided we have functions, that are able to read simple data.

File format can be defined by higher layer (acquired from config), but it is also possible to have parser, that knows only one file format. Such parser in our design is `SingleValueParser`, that does not accept any parameters concerning format.

The main job of the parser is getting information about file, so it can provide list of all sensors, that were found in the file.

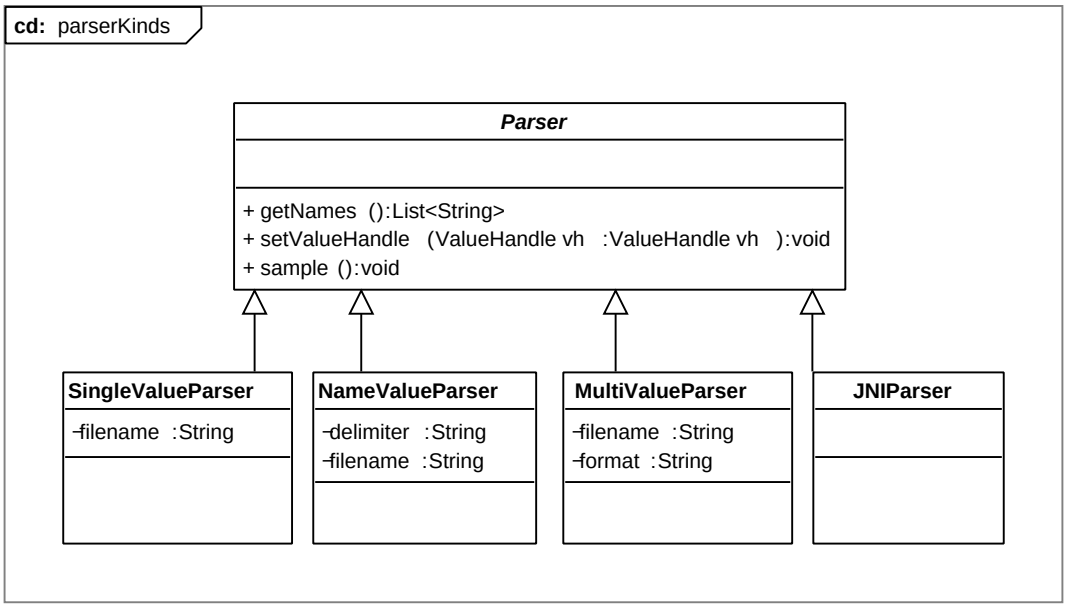


Figure 3.2: Design of parsers on logic level

We have projected parsers to provide easy extendability. If someone will want to extend the framework for support of new file types, he can write only a new parser to existing probe[1]. Without the parser, one would have to rewrite parts of the probe or readers, that does not sound scalable nor extendable.

3.1.4 Readers

Readers are lowest laying parts of whole stats gatherers, that will be programmed by us.

The main idea behind readers is to give programmer bigger parts, that could be used during parsing and that would provide partial data based on a request.

[1]The best probe for this job is `MultiValueProbe`, as discussed in Chapter 4.12

Such approach suggests code reuse, that will reward user with faster running code, because there will be less code used and less code means higher possibility of remaining in cache.

Another efficiency improvement is a possibility of reader reuse. Therefore, one parser is not required to have instantiate more readers for reading character sequence, that is readable by one reader. The main key in this optimalization is, that `OctetBuffer` holds its position independently and therefore one buffer allows being used by multiple readers.

To improve efficiency even further, we have decided to provide function `skip()`, that should be able to skip characters. Such skipping can be done more effectively by `OctetBuffer` and it also spares time without other support, because it does not have to construct return objects.

Responsibility of readers is reading basic elements from prepared `OctetBuffer`, without ability to create or fill such buffer. Readers will provide functions `readString()`, `readInt()`, `readLong()`, that will return `String`, `int` and `long` values depending on function called. Moreover, they will provide function `skip()`, that will skip the same count of characters, that would be read by other reading functions. Advance in `OctetBuffer` during reading any Buffer should be the same, regardless of reading function called, except when Exception in thrown to publish error in reading.

Readers are not obliged nor everytime expected to return all characters that were read, but should return the nearest value, that satisfies conditions given by reader type. Therefore, it is absolutely legal and should be expected, that reader returns first string encountered after newline or second encountered number, but these values should be returned in the order, in which they are found. Furthermore, programmer should be notified about exact behavior of the reader.

Readers should not reset their position in buffer, because it would affect reader stacking. Only looking forward is allowed thru `OctetBuffer` function `peek()`, that provides the very next character to be read.

Such design should allow stacking multiple readers without worries of getting different results, when `skip()` will be called instead of any different function. Stacking is calling multiple readers using same buffer consecutively.

Readers, in general, do not know and should never know, what is the meaning of all values in the buffer.

Using this possibilities, it should be possible to implement readers, that are conditioned by next characters, or even readers, that can read everything till endline or similar. Some examples implemented in this work are shown in Figure 3.3.

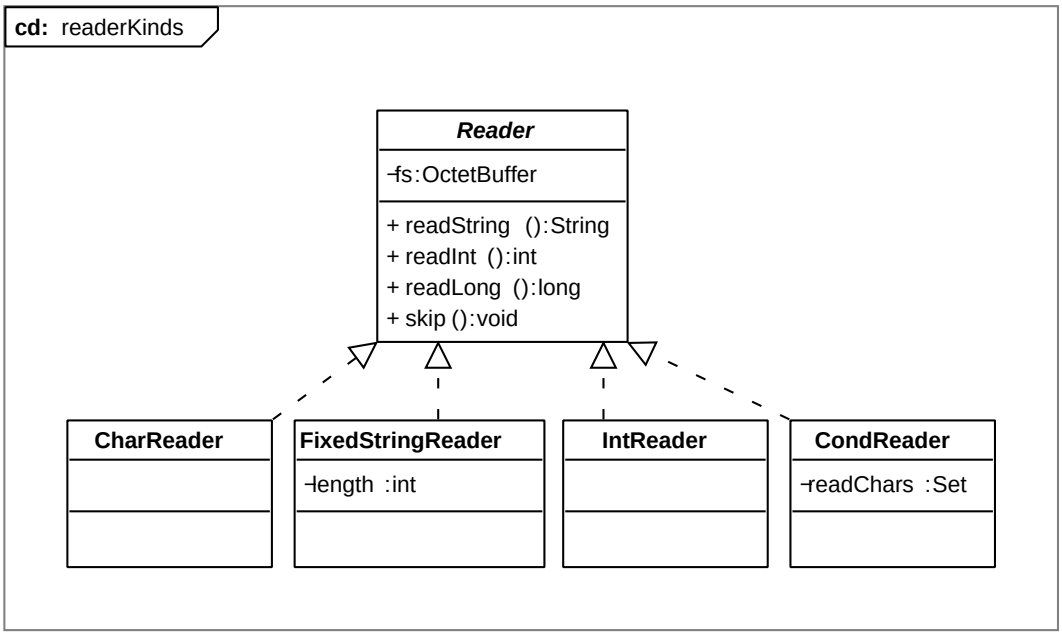


Figure 3.3: Design of readers on logic level

3.1.5 Octet buffer

Octet buffer is the lowest part, that is already provided by the framework and is able to read files.

Files in virtual file systems are in simple ASCII, while Java is Unicode-based and therefore does save twice as much data, when reading the virtual files.

Solution, that is already provided by the framework as `OctetBuffer` is able to read ASCII files without the mentioned overheat. It provides simple calls for advancing in the buffer and resetting position to the begining.

We will not analyze nor always mention it further, because it is only a part of framework, that is not designed by us.

3.2 Native data source

Native data source differs from file data source, because it is designed to be a layer between native code providing data and the remaining of JPMF, rather than thick layer providing sensors based on configuration. Configuration of sensors can be omitted, because all changes of native sensors require modifications of native code and more code changes therefore do not mean huge increase in complexity. Although there will be some changes, we want to leave configuration of data source descriptors. These are only user-visible labels, that are not really necessary for data source to work, but provide better user experience when selecting appropriate sensors.

3.2.1 Differences from file data source

We have mentioned in Analysis, that we will also use native parts, that will get values in their binary form. One design problem is, whether we should make

readers for these native functions. We have seen, that separate readers with defined reader interface would be possible to make, but additional interface would mean only unnecessary layer made just to be compatible. This layer would have to cache data received from native function – we want to traverse JNI as less as possible and one array copy is a way to go – and return elements one by one. Other way would be going thru JNI with every value. Both ways add resource usage without obvious advantages and because of it, we decided to make parsers with common interface to native readers, which will be used as any other parser.

Interface to native readers differs, because its main goal is providing multiple values to reduce overheat of JNI. Next reason to make different interface are resources, that would have to be used, if we wanted to get descriptions of provided values in any way. Static, hardcoded sensors are possible only because we do not expect configurability, as we described in Analysis.

Another advantage of different inner workings is, that native code does not need to be called, if measurement itself is not required and sensor is called. Such situation happen, when user wants only description of sensors, that is possibly done during every run.

Design of traversing through JNI is illustrated in Figure 3.4 - AbstractJNI-Parser should group all requests to provide same traversal of JNI across Parsers. On the other side of AbstractParser are native functions caring for JNI traversal and redirecting whole communication to pure C functions and correct readers.

Shared part of the native program used as a glue primarily takes care of descriptor allocation and mapping from common number - descriptor to memory required by parsers.

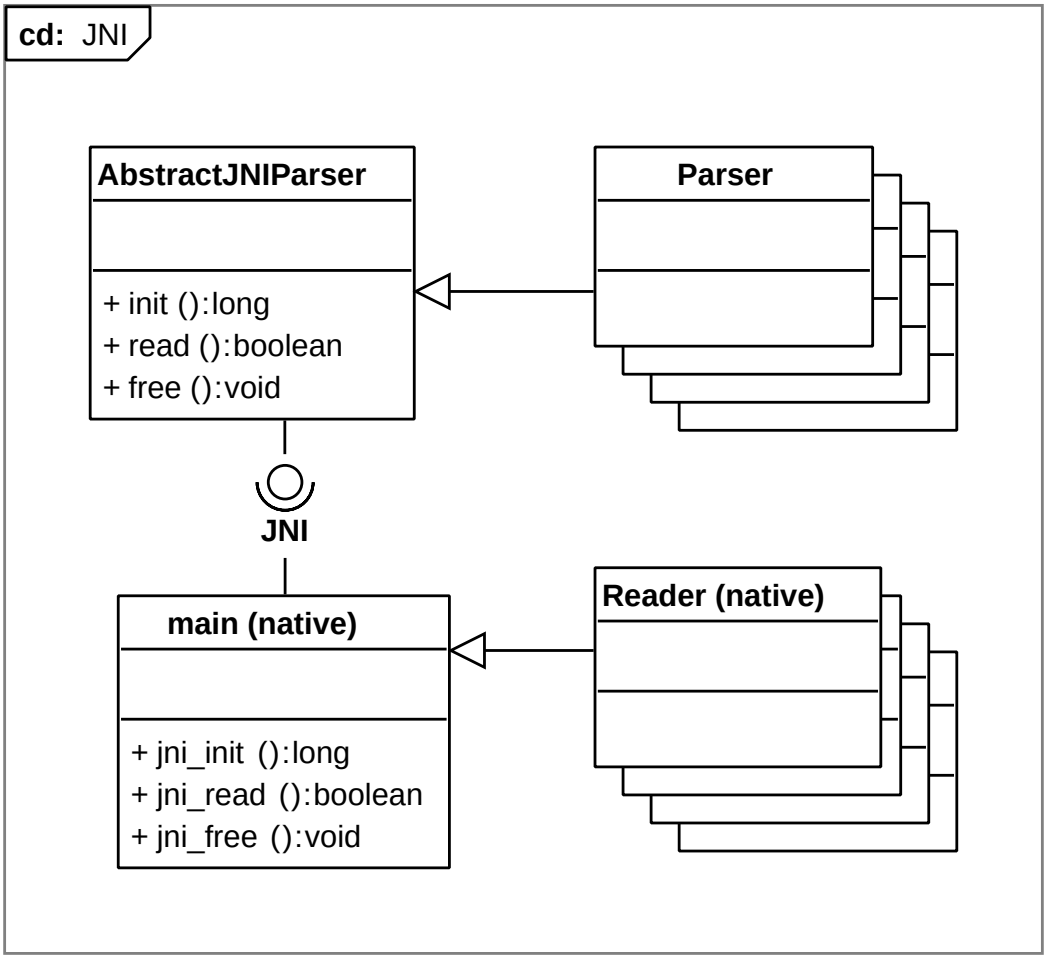


Figure 3.4: Design of native readers

3.2.2 Abstractness of native interface

One necessary design decision was, whether we should make one native interface for all parsers or more specific interface for each parser.

More interfaces could improve communication and allow more parsers to be created without modifiing native code. It could also reduce overheat connected with mapping all calls to one interface.

In spite of named advantages of multiple interfaces, we decided to make one interface for all native parsers. One interface is easier to understand and to support under Java, that is enough as an exchange for some overheat. Mentioned disadvantage of harder expandability is not as important as it may seem - we were not able to find, what could be expanded and even if a new method will appear, it is not much harder to bind it to custom parser.

3.2.3 Indication of measured data

Some native interfaces, such as that provided by Netstats interface, allow getting only some parts of performance data - only one stat at time. Reducing amount of syscalls to be done during sampling and amount of data exchanged with kernel could reduce system fingerprint on system resources and and such, improve

accuracy of measurements.

We have chosen array of booleans, that contained "false" for every sensor, that was not supposed to be measured and "true" for measured and active sensors. This array was sent to native code during initialization to reduce overheat.

We decided to benchmark sensors before and after insertion of this indication to prove improvements of this solution. We have measured time required to get 10 000 samples 100 times in a row, without time to construct parser or initialize native code[2].

After implementation of the new feature, repeated measurements had shown, that our hypothesis of faster sampling was absolutely wrong. Our tests show, that sampling all possible sensors in `NetStats` parser is with 95% confidence faster, than sampling with array full of "false" values - indication nothing is going to be measured, but with full initialization as required when sampling at least one value.

Results do not have obvious reason and are totally unexpected. Only viable reasons are cache aliasing and very fast syscalls after preparation (that was called in both measurements), that cause syscalls to be faster than reading value from program memory.

Based on the results, we decided to remove any indications of values, that should be measured.

3.2.4 Independence of the native code

During making native functions, we were not sure, which level of independence should our native code get.

One option is making completely independent program invocable from command line together with additional JNI glue. This would not be absolutely bad, as user would get isolation of two processes, that could communicate only by native and standart input and output. The main advantage of such design is really strict protocol, that could be easily independently tested without one side. Contrary, the most notable disadvantage is higher resource usage caused by parsing messages sent through the input or output pipe.

Therefore, we were thinking about different approach - partial integration. In this approch, all vital functions will be programmed independently from JNI and therefore easily testable, while there will be an interface to provide data to Java. Such approach has advantages in possibility of testing, although such testing needs more native code. Integration, that is not very close also brings one disadvantage – independent functions have to do their own memory allocations and data representation conversions once when getting data from kernel and again when providing it to Java.

There is also the last solution – making application closely bound to JNI and Java, that reduces overheat of copying and data conversions, but in exchange for simplicity of debugging.

We have decided to select the second option, because it does not lay in extremes. We will get relatively easy debugging while reducing overheat, if possible.

[2]Results are provided on attached CD

3.2.5 Accessing netlink

As netlink is a system interface, we can access it directly or use a library covering low level details.

Direct access without libraries provides simplicity and well documented interface. Documentation is not only encountered via official channels, but also Linux kernel sources can be used as an documentation, that is always up-to-date.

On the other hand, we have libnl, its sources and documentation. Its main advantage is, that it makes life of a programmer easier by counting bytes, if it has to be noted in a packet or by automatically padding neccesary parts, that speeds up whole process and reduces amount of bugs, that are hard to find.

All advantages of both sides, have almost equal weight. The ultimate feature, that was not mentioned and which decided finally used method is a simple access to debugging output provided by libnl. Upon request, that is made by setting one environment variable, `libnl` provides all communication, sent and received. It could be also easily made by any user, but `libnl` provides even more - message header is nicely parsed to the matching structures, while attributes and padding are also parsed and divided[8]. Therefore, debugging does not require look-up of characters by their hexedecimal value during reading Linux kernel source. Setting one variable works better instead.

Finally, we have chosen `libnl`, not because of advantages during usage, but as a result of possible simple debugging. However, if reading data will be performance critical, then this way can be deprecated and native code can be extended by methods, that will allow direct access to Netlink.

3.3 Sensor identifications and their meanings

Sensors have multiple identifications, where part is read only by users, part only by framework and there some identifications, that can be sometimes read by users, but could be read by framework.

Sensors identifications read only by user are sensor name and descriptions, that are provided only for information purposes, but are ignored by framework, when they are not provided to user.

Identifications read by framework are restricted on internal sensor ID. This ID is not provided directly to user and is at least remapped and as such, this can be virtually anything.

Remaining identifications read by users and framework are value type, that describes size of a space, that should be prepared for acquired values, and value kind, that describes, whether acquired data will be counter or gauge[1, p59]. `Gauges` provide current status of resource in time of measurement. Such value could be CPU or harddisk load or length of their waiting queues.

Contary to this, `counters` provide values, that can be expressed as an increment from one fixed time in the past. Basic requirements layed upon counters is, that they cannot decrement in the time, except during overflows. Such value could be "IO operation count since boot" or similar.

3.4 Sensor and instance naming

As an outer observer, we can see nothing more than grouped sensors within any datasource. We have already mentioned the neccessity of uniform naming across different systems.

Its solutions is Virtual Data Source[1, p59]. Virtual Data Source is a special type of data source, that does not own any probes. It gets data from other data sources and offers them under its own name. When all applications prefer Virtual Data Source, then it is possible to have uniform naming across different operating systems and even architectures.

But also if we have access to advantages of Virtual Data Source, we have to think, how to name our sensors to reduce time spent by mapping in Virtual Data Source and its configuration.

The easiest way to do this would be direct mapping of data gathered from file to its path, where the path separators are replaced with dots. Sensor name would be generic string or directly acquired naming string from `NameValue` files. Although this is a good solution in terms of configuration simplicity, as everything would run without user caring, this doesn't help with our requirement to have uniform naming across different systems and it would be slow, because traversing whole sysfs and procfs when we providing list of sensors could take long time and take resources to provide sensors, whose values will not change. Additionally, as Linux evolve and new modules are created, the files in VFS change[7, Documentation/hwmon/sysfs-interface]. Therefore, when using this approach, we are not able to be consistent within the system updates. Naming wouldn't change a lot, but change would happen even without warning user that something has changed.

The other extreme would be extension of the previous way, which would provide a way to give a pair for each sensor instance. That pair would say, which sensor maps to which public name. This would provide ways to configure it in our way and it would also give warning (it would not work anymore), if something internal would change. The main disadvantage would be the neccessity of adding every sensor instance to a configuration file. Imagine we bought new HDD and we would like to return performance data related to it - that would mean we would have to add one instance to each sensor, that makes tens of changes in this file.

We would like to provide a way to configure this without mentioning HDDs so many times as many HDDs we have. We decided to make it simple - we will provide `variables`, that will be able to have more possible values, will have support wildcards and will be usable everywhere, where a path is expected. When the path will contain a variable written in a special format, its value will be used as name or instance of sensor.

Sensor name retrieved from file contents or a file name does not provide a way to access data in a platform-independent way, and thus user cannot be satisfied with such sensor and instance names. To give user an ability to change sensor name as he wishes, we want to provide a way to get mapping to configuration and this mapping to be respected. Mapping should provide renaming sensors, for example from sda1 to hdd1. It seems a good idea to implement it using regular expressions, because it is well known among users and strong enough to provide

any remappings.

Eventually, we will get naming, that is not cross-platform, but is easily customizable and provides an easier way to make Virtual Data Source, that will rename all sensors and provide cross-platform naming.

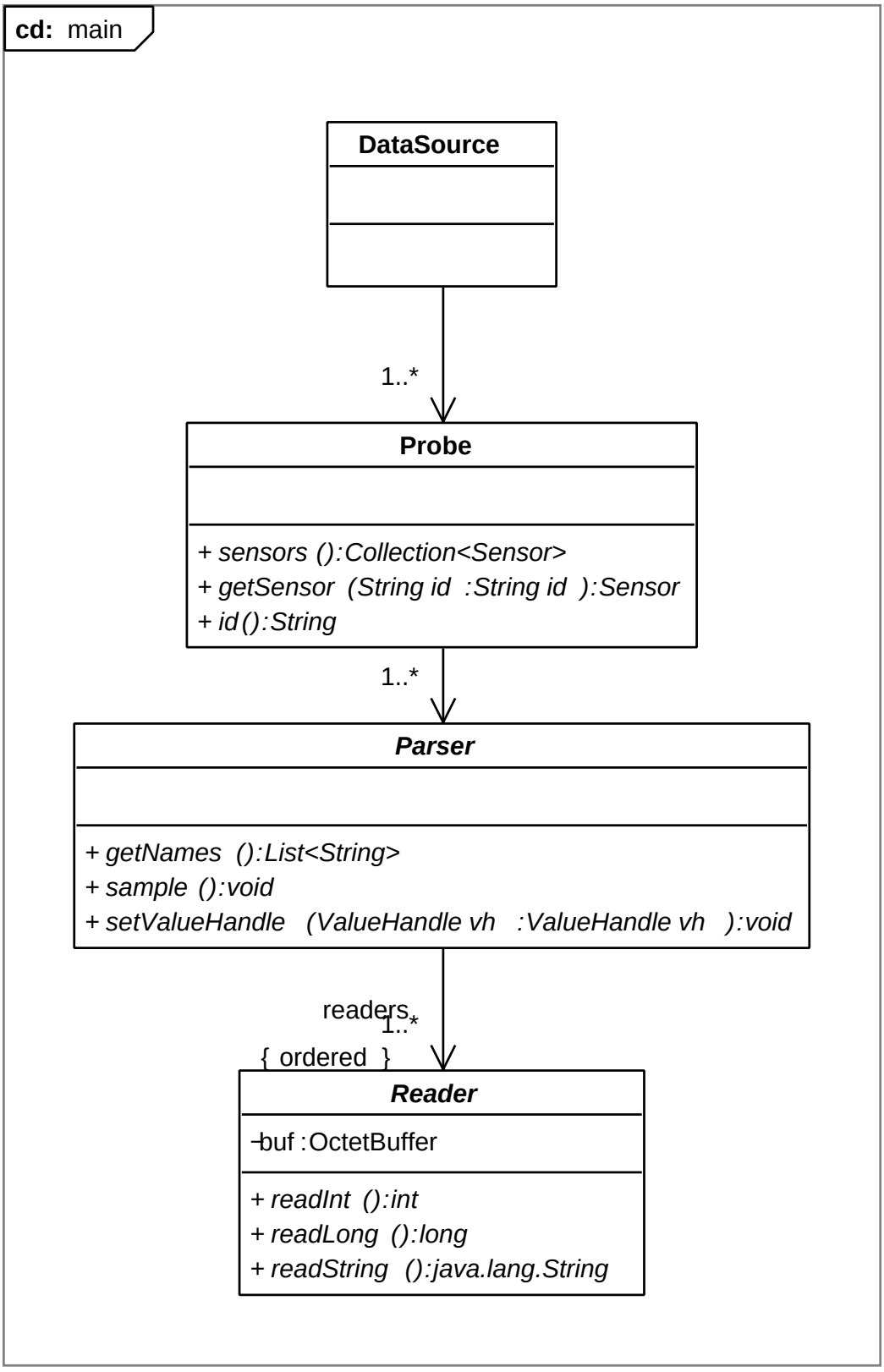


Figure 3.1: Basic architecture without details

4. Implementation

In this chapter, we describe implementation itself with some implementation details and decisions, that were not important enough to analyse or design them.

We have implemented a small GUI to test our work. Small GUI can be used for data source inspection, to enumerate sensors and their descriptions and to create `MeasurementContext` using these files and to sample data. Therefore, GUI can test, whether all interfaces are used correctly and can visualise status of structures, that are exposed.

4.1 Readers reading from files

We have designed an interface `GenericReader`, that should be implemented by all readers. `GenericReader` is described in Table 4.1.

We have noticed, that although we have projected multiple different reader types, all readers required by available file types can be implemented by three generic classes, that can be branched later.

Mostly branched reader is called `CondReader`. User provides it characters, that should be read and reader reads them, while stopping on any character, that is not in the provided list. This reader was used to build `LineRemainingReader`, that reads any values till and including newline represented as
n. `CondReader` could read any file from virtual file systems, but it is not optimal for simple specification of format and for the speed of parsing.

As a solution, we have made `FixedStringReader`, that reads strings or number of fixed length. This possibility speeds up whole parsing, because some parts of virtual files can be skipped without inspecting their elements and without lookups, what should be done with characters already read. One special reader made to be simple is `CharReader`, which can read one character of any type.

Because numbers do not have structure consisting of only same character, we have continued with `NumberReader`. Such non-uniformness can be seen in minus sign before the number, whereas minus sign cannot be read if it separates more numbers. `NumberReader` provides only reading of the numbers with the possibility of skipping them.

`readLong()`	gets next integer from the current position in associated buffer
`readInt()`	like `readLong()`, but returns shorter result
`readString()`	reads String from associated buffer, where begining and legth of the string are a subject of reader type
`skip()`	skips the same amount of characters, that would be read using reding function

Table 4.1: Functions in an interface of all readers - `GenericReader`

4.2 Readers getting data by using native code

Readers, although mentioned in architecture, are not provided in Java. Data from JNI are harvested using native readers, that are united with parsers. This decision has been taken to reduce overheat connected with harvesting data from JNI parsers and by introduction of a new interface. Multiple traversals of JNI could be slow, thus the main idea of interface was a reduction of traversals, where one Java-native interraction through array copy is enough to provide whole native data to Java.

4.3 Parsers of data from native code

We have designed JNI parser interface to be universal. Reasons of selection one interface are described in Chapter 3.2.2. Interface is described in Table 4.2 and saved in `AbstractJNIParser`, that should be extended by every provided JNI parser.

`jni_instances`	returns instances of specified type - this can require calling native code and therefore, type is required
`jni_init`	initiates reading and the inner structures of given type; returned handle can be used in other functions requiring handle. Parameters can be used in any way - they are not checked externally
`jni_read`	reads data from specified handle and saves them to return array given during construction
`jni_free`	frees data associated with handle; handle cannot be used anymore.

Table 4.2: Functions used by JNI

We have build multiple native parsers, that are described in Table 4.3. First column describes parser codename - it is an internal name of class within Java and also naming of sensor in native code. Second column describes values, that can be expected, when user decides to use the parser.

`getrusage`	values returned by `getrusage()` syscall
`netstats`	network statistics acquired using netlink interface
`sysinfo`	values returned by `sysinfo()` syscall
`taskstats`	stats about current task provided by netlink interface; these stats require more privileges, as discussed in Chapter 4.10
`times`	values returned by `times()` syscall

Table 4.3: Native parsers codenames and returned values

4.3.1 Types

Initialization and call for instance names is based on the same enums of `JNIType` in C and Java code. Communication takes place in numeric values, that is guaranted to be same between C and Java, if the order of fields is same.

We have decided not to use different communication, because in any case, names or any parts have to be maintained in the same state across two programming languages and `enum` provides everything needed.

4.3.2 Handles

We have designed handles to save specific session data. Every handle is a number, numbering begins from 0, but users should not rely on the numbering. On the native side, each native parser can save one pointer to the handle by returning it from init. Negative handle means an error during parser instantiation or freed handle.

4.3.3 C functions

We use uniform naming within our C functions, binding to Java code is done semi-automatically by adding functions to one switch.

```
void * <name>_init(const char **params, int paramlen, bool *success);
long * <name>_read(void *hdl, size_t length);
void <name>_free(void *hdl);
```

Where <name> is the codename of actual parser as mentioned in Table 4.3. Initialization function takes parameters to help parser instantiation and a parameter used to return, whether success was encountered. Returned pointer is any pointer, that will be later given to the function as hdl on reading and freeing.

Reading function gets mentioned pointer returned from init and a length of data, that should be read. Length is provided from the length of return array, that was passed from Java on initialization. This length is expected to be acquired by a side channel, as well as value naming.

And eventually, freeing takes pointer received on initialization and frees resources, that were allocated. Parser cannot be used afterwards.

4.4 File parsers

Parser are realised in the exactly same way as it was suggested in the Chapter 2.5.1. We provide `SingleValueParser`, `NameValueParser` and `MultiValueParser` to suit all needs.

4.4.1 SingleValueParser

`SingleValueParser` is the simplest parser, that is unique in one way – it is the only parser, that gets the name of value to be read directly from class constructing this parser. Such idea was implemented to be compatible with other parsers, that provide correct names of values, that will be read from files.

4.4.2 NameValueParser

Parsing using `NameValueParser` is pretty easy - it gets delimiter, `OctetBuffer` and is ready to sample. Internally, we get names of values from file once and we

just skip them during performance-critical reading. Delimiter can be also skipped, but we have prepared checks to discard possibility of changing line lengths.

4.4.3 MultiValueParser

When user requires entering file format, or file format is not one of the two described formats, then it is the best time to use `MultiValueParser`. Format is provided as a simple description of file contents. Format string begins with format specifier and if value should be also provided to framework, than is this followed by name in curly braces. These format specifiers can be put one after another or can interleave with strings, that will be otherwise skipped. Currently, we provide basic format specifiers found in files, that are described in Table 4.4.

`%d`	read number
`%b`	read all whitespace
`%n`	read everything till newline with newline
`%%`	% interpreted as a text

Table 4.4: MultiValue file format specifiers

Therefore, correct part of format specifier is `%d{hdd.reads}` for decimal value containing reads of harddisk or `%d` for decimal value, that should be skipped.

4.5 Probes

4.5.1 File - related

We have implemented probes to read exactly the file types, that are readable by parsers. They generally get list of files to parse together with instance provided by with each file. `NameValueProbe` gets its delimiter, `MultiValueProbe` gets a format string. Furthermore, probes are provided with `SensorConfigGetter`, that is initialization-time provider of config, that is able to provide descriptions and mapping for requested instance.

All three file probes are extensions of `AbstractFileProbe`, that provides all necessary services. Almost all services can be configurated by providing settings in `SensorConfigGetter` and by setting one internal map providing descriptions – `__sensorToInstancesFiles`. Its main goal is mapping a sensor name to its instances and files belonging to instances.

Therefore, probes are configurable by an external configuration, that is passed only as a parameter from a data source.

Probes related to files are made to be instantiated by ProbeBuilder.

4.5.2 Native

There are also native probes, that are all extensions of `AbstractJNIProbe`. Basically, all native probes are only wrappers around their respective parsers, that provide values further. The only exception in user's ability to configure Probe is in `SensorConfigGetter`, that is used to specify human readable descriptions.

4.6 Data source

We have implemented two data sources, that are the highest part of our hierarchy. Data sources are created by inheriting from `AbstractDataSource`, that is easily configurable itself. As data source is only a concentrator of sensors, everything can be configured by changing two look-up maps. One is `__mapGroupSensor`, that maps group name (that is also a probe id) and sensor name to sensor id. The other one, `__mapToInstances`, is maping from group name and sensor name to a configuration of human readable descriptions. There are no more configuration than this two fields.

`JNIDataSource` is made to be simple. It provides one static array, that contains classes to be instantiated and used later. All these classes are probes, that were mentioned as native probes.

`FileDataSource` holds all probes, that get their values from files. It gets configuration itself to get names of files, their instance names and descriptions.

Both `JNIDataSource` and `FileDataSource` have their own `DataSourceProvider`. One is `JNIDataSourceProvider` and the other is `FileDataSourceProvider`.

4.7 Configuration

Configuration and some reasons to create it was described in analysis in Section 2.3. Now, during implementation, new ideas appeared and would be nice to be implemented.

Configuration file is every file with XML extension located in the directory `etc/probeconfig.d` under the root directory of JPMF. Because our XML namespace is flat, we accept all configuration in single file or almost every directive to be in different file with any position in between. Every configuration file should contain a definition of used XML schema to allow validation. XML schema is located in `jaxb/res/namespace.xsd` under the root of JPMF. Such declaration is

```
<?xml version="1.0" ?>
<a:configRoot xmlns:a="http://jpda.dsrg.ow2.org/config.xsd">
```

or any similar with different namespace.

4.7.1 Basic elements

Configuration allows elements, majority of which has to be under the root element.

Variable is specified by element `variable`. All variables have to be specified in an order, where variables are used only after they were declared. Variables in paths or definition of other variables are enclosed in curly braces.

Example of declaring variable `variableName` with possible values inherited from `otherVariable` concatenated with `/fileName1` as the first value and with `/fileName2` as the second value:

```
<variable name="variableName">
<value>{otherVariable}/fileName1</value>
```

```
<value>{otherVariable}/fileName2</value>
</variable>
```

Configuration allows specification of file probes. It requires group, that will be used in URL-like description of sensors, followed by identifier, name, description (all values are provided only for human), possibly followed by mappings (as described in Chapter 4.8 and 3.4) and path to the file, which will be read. Note that more path can be sipmly accomplished using variables.

Real world example:

```
<single-value group="state-times">
    <identifier>StateTimes</identifier>
<name>State times</name>
<description>Times spent in CPU states</description>
<path>/sys/devices/system/cpu/{i:cpuname}/cpuidle/{s:stateTimes}</path>
</single-value>
```

Here, earlier defined variable `cpuname` becomes instance name and variable `stateTimes` becomes sensor name.

4.7.2 Variables in configuration

Because variables are not tied to the remaining of framework, we excluded them to custom class `VariableMapper`, that gets all variables and then translates between format with variables to names of existing files, where variables are not mentioned.

4.7.3 Changing native parsers

Code modification is an only way to extend Linux native parsers in JPMF. There are also other parsers and probes, that could be extended by code modification, but they are not, by concept, built to be extended so.

If we want to add parser, we should extend our native interface – we should add new types to native code and make same changes in Java.

4.7.4 Reading configuration

We have decided to use XML as a format holding configuration, that allows validation even from outside of our framework. Now, we have to decide, how we will read XML.

We are surely not going to make any custom parsers, because such implementation would take too much time and would not lead, in our opinion, to valid results.

We though about parsing configuration using SAX[11]. Its main advantage is its simplicity, that also causes disadvantages. We have to hold context and to save values, that were found. In exchange, we get calls of our functions, that allows simple parsing without big performance overheat and memory usage.

Parsing could be done using DOM[10], that is supplied with Java like SAX and provides thick layer on the top of XML. DOM can be called to get specified elements in a specified context, which reduces amount of code, that has to be programmed. On the other hand, there are disadvantages of higher memory

usage and computation complexity of getting elements, but these disadvantages are not so important, because parsing files and reading values runs only once during initialization.

Besides named disadvantages, reading configuration using SAX and DOM is not done in easily alterable way, and thus requires multiple code changes, that come with small changes in XML. This disadvantage can be solved, if we will use binding instead of parsing. Finally, we have chosen JAXB[9], that makes binding between XML data and Java classes. This approach requires only XML Schema[12] to generate classes, that can be followingly accessed as a part of tree in tree-like structure. Writing XML schema cannot be avoided, if we want to get automated schema validation and thus adding XML Schema does not add work to programmer.

We have separated the reading of configuration by our own interface, because of a probability of changes in configuration, that will stop in our interface instead of "infecting" whole framework with changes.

We decided to provide user one directory, where he is free to place scripts, that will be loaded without any modification of Java files. Directory for multiple files gives user and configurator ability to add or remove new files as required, without altering files and fear of irreversible change.

Designed XML format is simple - basic configuration goes first and general information afterwards. Separation of configuration to different files is optional and without obstacles in way, which is a major advantage in terms of readability.

4.8 Renaming sensors and instances

In Chapter 3.4, we have provided reasons of mapping and gave possible solution of using regular expressions provided by user to get mapping specification.

During implementation, we noticed, that one regular expression is not neccesarily enough and therefore, we have implemented chaining of regular expressions – they are executed in row, as they were entered.

Because there are sometimes multiple values to remap and remapping cannot be described using generic regexp, that leads to one-to-one mapping, we have provided a option to specify pairs describing, which value will remap to which result. Such remapping could be done by regular expression, but a user would have to escape all special characters such as dot, that is easy to forget. With pairs, there is no need to escape characters and whole implementation is certainly faster, because no regular expression is constructed and strings are compared using generic way provided by Java.

Mapping is specified in the configuration by an element `mapInstance` or `mapSensor`, depending on the type of mapping. Both mappings have same possibilies, so we will describe then in a row. Before begining, we should note, that mappings stack in the order they are specified. Therefore, it is reasonable to specify more mappings in a row.

Most important mapping is `regexp`, that lies in the element of same name. Two subelements `from` and `to` specify original sensor or instance name and their mapped versions. Matching using parentheses is also allowed; references to old values are done using dollar with number of the match.

Another mapping with subelements `from` and `to` is described by an element `pair`. Such element maps one value to another, without changing any different values.

Last two mappings are `prefix` and `suffix` mapping, that add prefix or suffix to string received. Such action could be reached by writing regex, but this is faster and easier, so we have provided it.

4.9 Saving C integer to Java integer

Java supports signed integral values of size 8, 16, 32, 64 bits and numbers of unlimited size (with some overhead) using BigInteger[4, p492]. All of these sizes are given and cannot differ, even if we use different compiler or runtime environment.

Type	Java size	C size	C size2
short	16	16	16
int	32	32	32
long	64	32	64
long long	—	64	64

Table 4.5: Comparison of datatype sizes in bits

On the other hand, C used in Linux kernel provides datatypes, such as int, long and long long, without noting, how many bits it should occupy. We have made a table 4.5 using one Java compiler and one C compiler (`gcc`) run with different parameters, that forced 32 bit and 64 bit compilation. This table demonstrates, that even same compiler produces different sizes of data types depending only on configuration and we cannot rely on same size of given types on an implementation level with same compiler.

Only thing we know about data types and that is given in specification, that they are ordered in size. We have constants `INT_MAX` and `INT_MIN` (and similar for `long` and `long long`), but this does not help in our Java application. We could try compiling extra native helper, that would be loaded by Java, but this would require using same compiler with exactly same parameters to that used during kernel compilation. This could be possible for users, who compile their kernels themselves, but it would take too many time or would be close to impossible for users, who use kernels provided by their Linux distribution.

There is also a possibility to use longer datatypes, in case of uncertainity, that can hold even smaller values as an exchange for space taken. As a result, we decided configurator will choose, what are datatypes of target systems.

4.10 Missing privileges while using netlink

Netlink interface provides detailed and exact per-process data, that are only accessible for users with `cap_net_admin` capability (as specified by withdrawn Posix 1003.1e). Main reason for this restriction is a possibility of sensitive information leak.

Having access to this data, one could get the count of read characters and easily compute password length[6]. This is the main reason, why we will not try bypassing this protection (eg. with setuid binary), but will use it.

There are more ways to solve missing privileges.

The simplest is, obviously, not to solve it. Using this solution, data will not be available, when user will run an application with insufficient privileges. Although it may seem the worst solution, when JVM is run by user without enough privileges, it has also advantages. Firstly, the most notable advantage is higher security. There is no possibility to make a security vulnerability leading by exposing higher privileges, when our program won't have such privileges. There is also a possibility of dropping privileges as soon as we will be allowed to initialize, but this is much more violent action than it would seem. If JVM would be run with elevated privileges with different reason than getting more exact data, we would, actually, drop that privileges without user knowing it and could cause, that another part of program will miss it.

The other way to solve it is a separation of library requiring rights to different executable and so to different process. It could get the necessary privileges, when being run using setuid bit. Unneeded privileges could be dropped as soon as possible without leaving big security hole behind. All processes would get only information about themselves (parent process of process with partially elevated privileges). It would, however, require fork & exec from JVM process. Linux uses only lightweight fork, but it still fails, if process takes more than half of `free memory size * overcommit_ratio`. This means, forking and initializing our probe could fail "randomly" (for external observer), if the JVM took around one half of total memory.

One possible way to solve fork memory problems is running custom netlink proxy (outside JVM) with higher privileges. This proxy cannot only simply resend everything received with higher privileges as everyone could send such requests and get access to sensitive information of another process. There is a possibility of making whitelist, but an attacker could still get sensitive info about other processes. Another solution is moving whole netlink communication logic to proxy. Such proxy could check PID of request sender and would construct a request with this PID. Checking PID of sender requires such amount of resources (at least opening some files in procfs), that we will not do it.

As a solution, we decided to solve whole problem in an easily programmable and easily runnable manner. Library will not drop privileges, unless compiled with special parameter, that will ask it to drop privileges.

4.11 Changes to the framework

JPMF provides well designed datasource interface, but one enumeration had to be extended to express meanings of returned data. We have also discovered, that framework does not behave as it was probably expected and corrected behaviour.

4.11.1 Unsigned values in Java and JPMF

Unsigned integer datatypes are not supported by Java, nor by JPMF, but appear often in counters associated with Linux kernel.

These counters are integers, that are usually 32 or 64 bits long, but the length is not guaranted to have any special bit length.

Our first idea was giving larger datatype to unsigned values. When used with at leas 32 bit C datatype, we would have to use 64 bit long provided by Java. This would cause doubling memory consumption and disallow reading unsigned 64 bit integer from C, as there is no bigger integral type, if we do not want to use ineffective BigInteger. This means, that we should look for another way.

Next and possibly the most effective idea is saving unsigned C datatypes to signed Java datatypes of same size. This could cause some problems, but fortunately, JPMF is prepared for such storage. Datatypes will be set by configurator, so we will not have to bother with them.

4.11.2 Operations on unsigned value to signed datatype

The question is, whether we can use signed data type instead of unsigned data type of same bit size without special operations used during calculations.

We do only 2 elementary operations during parsing data - adding and multiplication by 10. Adding works exactly same for signed and unsigned integers, if we use 2's complement, that is guaranteed by Java Language Specification.

Before discussing multiplication, we should think, when problems can appear. Operations with $(k-1)$ bits of k bit data type when using positive numbers are guaranteed to be same in a numeric way - both signed and unsigned number are same and are expected to give same results, as they represent one number.

Therefore, any problems with multiplication, can appear only during last multiplication by 10 — these numbers grow 10 times every time, when they are multiplied. In case numbers have same bitlength, most significant bit will be visited only during last multiplication by 10 (we know, that $10 \geq 2$). And result of this multiplication is guaranteed - it should be cropped to the size of datatype, no matter, whether datatype is signed or unsigned. This means, we can save k bit value by cropping to k bit signed datatype[4]. □

4.11.3 Signed or unsigned indication

Unsigned values saved in signed data type need any indication to let user (and also other programs) know, whether first bit is sign bit or should be interpreted as most significant bit of number itself.

The ultimate solution is a concept of ValueHandle provided by JPMF. ValueHandle has its type and storage, that are two different properties, so two different ValueHandles can use same type of storage. We will use this idea and add two new types of ValueHandles, that will be capable of saving unsigned values, but with same kinds of storage, that are provided for signed values.

4.12 Adding support for a new type of a file

There are multiple approaches, that can be used if we want to add support for a new type of the file, that will appear in virtual file system.

The easiest and the recommended way for almost all new files would be adding custom percent format specification in `readPercentType()` located in

`FormatParser`, which is the inner class of `MultiValueParser`. This approach is the simplest, because one change can be immediately used in configuration and described like any currently supported multi value file.

If there was a completely new type of files, that would also provide a naming for the values, one can write custom parser and probe

5. Conclusion

We have successfully implemented custom data sources able to read performance stats from multiple sources. It provides all interfaces suggested by JPMF and also an ability to extend it by altering human readable and editable configuration.

5.1 Future work

Our work is open to be further extended and there are many possibilities, where new ideas can be added. Part of the work should be focused on the maintenance of list of files, together with their types and descriptions. If such maintenance will be omitted, sensors will gradually begin to fail.

Next design goal should be making Virtual Data Sources, that will be able to provide uniform naming across different operating systems. It would be a nice idea, if it could work in a semi-automatic manner, that would use our remapping possibilities to make its configuration easier.

There are even more challenges ready for future exploration. One of them is making a kernel module, that would gather performance information from various places through whole kernel and would, as a result, provide detailed and also system-wide information centralised in one place. Provided data and interface could be similar to `Windows Performance Data` provided by Windows or to `kstat` provided by Solaris. Providing data itself could be done through recommended `Netlink` interface. Note that such kernel module would not need only extensive configuration describing getting values from various sources, but also userspace data source, that would connect it to JPMF.

Bibliography

[1] Bulej, Lubomír. *Connector-based Performance Data Collection for Component Applications*. Dissertation Thesis, Dept. of Software Engineering, Faculty of Mathematics and Physics Charles University in Prague, September, 2007

[2] Bulej, Lubomír, Marek, Lukáš *Java Performance Measurement Framework Manual*. version 1.0 Q-ImPrESS Consortium, 2010

[3] Dráb, Martin. *Extending Java Performance Monitoring Framework with Support for Windows Performance Counters*. Bachelor Thesis, Department of Software Engineering, Faculty of Mathematics and Physics Charles University in Prague, May, 2012

[4] Gosling, James. *The Java^{TM} language specification*. 3rd ed. Upper Saddle River, NJ: Addison-Wesley, ©2005, xxxii, 651 p. ISBN 0-321-24678-0.

[5] The IEEE and The Open Group. *The Open Group Technical Standard Base Specifications*. Issue 7. [online]. 2008 [cit. 2012-06-20]. Available from: `http://pubs.opengroup.org/onlinepubs/9699919799/idx/index.html`

[6] The Mitre Corporation. *CVE-2011-2494: kernel/taskstats.c in the Linux kernel before 3.1 allows local users to obtain sensitive I/O statistics by sending taskstats commands to a netlink socket, as demonstrated by discovering the length of another user's password.* "Common Vulnerabilities and Exposures: The Standard for Information Security Vulnerability Names [online]. 2011 [cit. 2012-06-20]. Available from: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-2494`

[7] Torvalds, Linus and contributors. *Linux Kernel 3.0*. [Computer program source] 2012.

[8] Graf, Thomas and contributors. *Netlink Library (libnl)*. version 3.2 [online]. 2011-05-09 [cit. 2012-06-20]. Available from: `http://www.infradead.org/~tgr/libnl/doc/core.html`

[9] Kawaguchi, Kohsuke et al. *JSR 222: JavaTM Architecture for XML Binding (JAXB)* version 2.2 [online]. 2009-12-10 [cit. 2012-06-20]. Available from: `http://jcp.org/aboutJava/communityprocess/mrel/jsr222/index2.html`

[10] Le Hégaret, Philippe. *The W3C Document Object Model (DOM)* World Wide Web Consortium [online]. 2002 [cit. 2012-06-20]. Available from: `http://www.w3.org/2002/07/26-dom-article.html`

[11] Megginson, David et al. *SAX JavaDoc* [online]. 2004-04-27 [cit. 2012-06-20]. Available from: `http://www.saxproject.org/apidoc`

[12] W3C. *XML Schema* [online]. [cit. 2012-06-20]. Available from: `http://www.w3.org/XML/Schema`

[13] Edgewall Software. *Munin monitoring* [online]. 2012-07-24 [cit. 2012-07-30]. Available from: `http://munin-monitoring.org/`

[14] Ganglia contributors. *Ganglia Monitoring System* [online]. 2012-07-16 [cit. 2012-07-30]. Available from: `http://ganglia.info/`

[15] Brodal, Gerth Stølting, Fagerberg, Rolf, Vinther, Kristoffer. *Engineering a Cache-Oblivious Sorting Algorithm*. 2004. Available from: `http://www.cs.au.dk/~gerth/slides/alenex04.pdf`

[16] Kupka, Martin. *Ministr trestal za registr aut. Vyhodil ředitele, dalším vzal peníze*. 2012-07-24 [cit. 2012-07-30]. Available from: `http://zpravy.idnes.cz/domaci.aspx?c=A120724_190041_domaci_mku`

[17] Knuth, Donald Ervin. *The Art of Computer Programming*. Volume 3: Sorting and searching. Addison Wesley Longman, 1998. ISBN 0-201-89685-0

[18] Intel Company. *Using the RDTSC Instruction for Performance Monitoring* [online]. 1997 [cit. 2012-07-30]. Available from: `http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf`

[19] OW2 consortium. *LeWYS* [online]. [cit. 2012-07-30]. Available from: `http://forge.ow2.org/projects/lewys/`

[20] Kalibera, Tomáš et al. *BEEN - Benchmarking Environment* [online]. [cit. 2012-07-30]. Available from: `http://been.ow2.org/`

List of Tables

List of Figures

List of Abbreviations

JPMF Java Performance Measurement Framework - big project, that is extended by this thesis

SingleValue type of files described in Chapter 2.2.1

NameValue type of files described in Chapter 2.2.2

MultiValue type of files described in Chapter 2.2.3

JNI Java Native Interface - interface between Java and native code, that allows calling of Java code by native code and vice versa

Attachments

Attachment 1: CD with JPMF

Attached CD contains software project together with its sources. The most important files are mentioned in Table 5.1.

`thesis.pdf`	This thesis with clickable references
`jpmf`	Whole framework, that was described in this thesis
`jpmf/README`	Details about compiling and running JPMF
`stats`	Informational stats of time taken (in μ seconds) to complete 1000 iterations of getting 100 (and 10)

Table 5.1: Basic contents of attached CD